Efficient Collision Detection for Brittle Fracture

Loeïz Glondu, Sara C. Schvartzman, Maud Marchal, Georges Dumont, Miguel A. Otaduy

Eurographics/ ACM SIGGRAPH Symposium on Computer Animation (2012), pp. 1–10
P. Kry and J. Lee (Editors)

Efficient Collision Detection for Brittle Fracture

Loeiz Glondu[1,2] Sara C. Schvartzman[4] Maud Marchal[1,3] Georges Dumont[1,2] Miguel A. Otaduy[4]

1-IRISA/INRIA Rennes 2-ENS Cachan, Antennes de Bretagne 3-INSA Rennes 4-URJC Madrid

Figure 1: ***Smashing Plates.*** *The user drops balls in real-time to smash the plates, and at the end of the simulation the scene consists of more than* 45*K triangles. The complete simulation runs at* 8*ms per time step on average, with a maximum of* 17*ms.*

Abstract

In complex scenes with many objects, collision detection plays a key role in the simulation performance. This is particularly true for fracture simulation, where multiple new objects are dynamically created. In this paper, we present novel algorithms and data structures for collision detection in real-time brittle fracture simulations. We build on a combination of well-known efficient data structures, namely distance fields and sphere trees, making our algorithm easy to integrate on existing simulation engines. We propose novel methods to construct these data structures, such that they can be efficiently updated upon fracture events and integrated in a simple yet effective self-adapting contact selection algorithm. Altogether, we drastically reduce the cost of both collision detection and collision response. We have evaluated our global solution for collision detection on challenging scenarios, achieving high frame rates suited for hard real-time applications such as video games or haptics. Our solution opens promising perspectives for complex brittle fracture simulations involving many dynamically created objects.

1. Introduction

Fracture is commonplace in crashes and explosions, essential ingredients of action entertainment both in feature films and in video games [BBC*11]. When objects fracture, multiple object fragments collide and pile up, making fracture simulations extremely collision intensive. The recent advent of fast algorithms for fracture crack computation [BHTF07, PO09, GMD12] has made collision handling a dominant cost in fracture simulation.

The simulation of fracture imposes two major challenges on collision handling. First, acceleration data structures for collision detection need to be created and/or updated at runtime, due to topology changes. Second, the newly created crack surfaces arise in parallel close proximity, which constitutes a worst-case scenario for collision detection and response, with many surface primitives in contact, less chances for high-level culling, and no temporal coherence. Offline animations may afford spikes in the computational cost at fracture events, with the cost being amortized over the length of the simulation. But in interactive applications such as video games, simulation must comply with a maximum computational budget per time step, calling for efficient solutions at all simulation frames, particularly at fracture events.

In this paper, we present an efficient solution for collision detection in brittle fracture simulations, suitable for highly demanding applications such as video games. Our solution exploits several observations about object behaviors and contact events in brittle fracture. Most importantly, brittle objects undergo little deformation before fracture [OH99], hence, for collision handling purposes, they can be treated

as rigid bodies between fracture events. Therefore, our approach to collision detection, outlined in Section 3, relies on well-known efficient acceleration data structures for rigid body contact, namely distance fields and sphere trees.

However, distance fields and sphere trees typically rely as well on constant topology, and suffer a heavy preprocessing cost. For their efficient integration in brittle fracture simulation, we propose novel algorithms for fast reconfigurable distance fields and sphere trees. In Section 4 we present a novel method to compute an approximate interior distance field for fracture fragments. Our method exploits features of fracture simulation and collision response algorithms to optimize its storage and computational cost. In Section 5 we present a novel sphere tree data structure, well suited for fast updates under fracture events.

To reduce the cost of collision detection and response due to close parallel crack surfaces, we exploit the observation that the majority of the contacting primitives defines redundant contact constraints. As a result, we propose a design of the sphere tree that lays the foundation for a simple self-adapting collision detection algorithm at run-time. It is executed as part of hierarchical collision detection, not as a post-process, thus enabling high-level pruning, and reducing the cost of both collision detection and response.

In Section 6 we evaluate our data structures and algorithms on several challenging fracture simulations, demonstrating very high simulation frame rates, suited for hard real-time applications such as video games, as shown in Fig. 1. We also analyze performances under various algorithm settings and object resolutions.

2. Related work

We focus our discussion of related work on the two main data structures used in our method, namely distance fields and sphere trees, on adaptive collision detection methods, and on collision detection techniques designed particularly for fracture simulations.

Distance Fields store in a grid distances to the surface of an object, and possibly the distance gradient. For rigid bodies, distance fields may be precomputed, hence the computation of penetration depth of a point inside a rigid body becomes trivial [GBF03]. Adaptive distance fields [FPRJ00] store distances in an octree to reduce storage requirements. In some applications it is even sufficient to store information only near the surface of the object [MPT99]. Distance fields have also been used for deformable bodies by fast recomputation [SGGM06] or by approximating finite-element [FL01] or modal deformations [BJ08]. In various applications of computer animation, distance fields have been approximated using front propagation algorithms [HTK*04] or graph-based distances [SOG06].

Sphere Trees are one of the classic types of bounding volume hierarchies for fast pruning in collision detection [PG94]. Kaufman et al. [KSP07] integrated adaptive distance fields with sphere trees. Fast culling is possible thanks to the sphere trees, while the adaptive distance field is used for accurate penetration depth queries. Weller and Zachmann [WZ09] designed inner sphere trees for the fast computation of penetration volumes.

Adaptive and Time-Critical Collision Detection. One interesting use of sphere trees is time-critical collision detection [Hub96]. The output of time-critical collision detection was later optimized by considering also collision response and adding adaptivity based on visual perception metrics [OD99]. Other ways to govern adaptivity in time-critical collision detection include error control based on potential intersection volumes [KZ03]. With contact levels of detail [OL03], adaptive collision detection is extended to arbitrary types of bounding volumes, contact points are computed using surface approximations at each level, and various adaptivity criteria can be considered. Yet a different approach to limit the cost of hierarchical collision detection in an adaptive manner is stochastic sampling [KNF04]. Barbič and James [BJ08] introduced adaptive time-critical collision detection to sphere-tree vs. distance field queries.

Collision Detection for Fracture. Acceleration data structures for collision detection need to be updated or recomputed at fracture events, because precomputed distances or bounds are no longer valid or tight, and new surfaces need to be considered. Larsson and Akenine-Möller introduced the concept of selective restructuring of bounding volume hierarchies, according to fitting-quality metrics [LA06]. Otaduy et al. [OCSG07] applied local restructuring operations to limit updates in progressive fracture. Recently, Heo et al. [HSK*10] have presented an algorithm that finds a good compromise between bounding volume restructuring and fast recomputation.

All these approaches suffer two major limitations for simulations of brittle fracture. First, the quality of bounding volume hierarchies degrades immediately under brittle fracture, and full recomputations are needed. As a result, large computational spikes decrease the simulation cost at fracture events. Such spikes can be amortized in offline simulations, but not in hard real-time applications such as video games.

Second, earlier research works on collision detection for fracture simulation [LA06, OCSG07, HSK*10] have typically focused on the problem of surface intersection, which unfortunately does not address the needs of collision response in rigid body engines. Robust and efficient rigid body engines in the video games and feature film industry rely on velocity level constraint-based solvers followed by stabilization or drift correction [Erl07, KSJP08]. These solvers need contact information in the form of penetrating points, distances, and directions, and our collision detection algorithm satisfies these needs.

Complementary to our approach, collision detection for fracture simulations can also benefit in several ways

from fast parallel algorithms executed on graphics processors: for fast culling in piles of objects [LHLK10], collision detection queries with no need for preprocessing [FBAF08], or fast computation of data structures, either distance fields [SGGM06] or bounding volume hierarchies [LGS*09]. We have designed methods that achieve high performance by reducing computations (i.e., they are less computationally demanding than previous methods), and could also benefit from parallel implementations.

3. Overview of the Collision Detection Algorithm

We execute collision detection tests between pairs of objects A and B, which may be either original unfractured objects or fragments resulting from fracture events. Without loss of generality, let us refer to them as fragments. We augment each fragment A with two data structures for collision detection: a distance field $D(A)$ and a sphere tree $S(A)$. Section 4 and Section 5 describe the contents, construction and update of our novel *fragment distance field* and *fracturable adaptive sphere tree* data structures.

When broad-phase collision culling returns the pair (A,B) as potentially colliding, we query both $S(A)$ against $D(B)$ and $S(B)$ against $D(A)$. The result of a query $(S(A),D(B))$ is a set of contact constraints C, each defined by a tuple $(\mathbf{c},d,\mathbf{n})$. $\mathbf{c} \in A$ is a contact point, d is the closest distance from $\mathbf{c}$ to the surface of B, and $\mathbf{n}$ is the contact normal. If $\mathbf{c}$ is inside B, d is negative and represents the penetration depth. After collision detection, we feed the complete set of contact constraints to a constraint-based contact solver with a velocity-level LCP (with friction), plus constraint drift correction. In our examples, we have used the off-the-shelf contact solver built in Havok Physics.

In our algorithm, a query $(S(A),D(B))$ builds on three elementary queries involving nodes of the sphere tree $S(A)$. Each node is represented by its center point $\mathbf{p} \in A$ and a radius r. Then, the three elementary queries are:

- insideTest$(\mathbf{p},D(B))$: it determines whether $\mathbf{p}$ is inside or outside B.
- penetration$(\mathbf{p},D(B))$: when $\mathbf{p}$ is inside, it computes the penetration depth from $\mathbf{p}$ to the surface of B, as well as a penetration direction $\mathbf{n}$.
- sphereTest$(\mathbf{p},r,D(B))$: when $\mathbf{p}$ is outside, it performs a conservative test for intersection between B and the sphere of radius r centered at $\mathbf{p}$.

These three elementary queries will be described in detail in Section 4. In all our descriptions, we assume that the point $\mathbf{p}$ has been transformed to the local reference system of fragment B.

Our collision detection algorithm, outlined in Algorithm 1, traverses a sphere tree $S(A)$ in a breadth-first manner, and prunes branches that are completely outside the other fragment B. Pruning is efficiently executed by comparing the radius of a sphere and the distance from its center to the surface of B. The algorithm can be easily modified to allow for a contact tolerance ε. A contact constraint is added to C if the distance is shorter than ε, and the query descends to the children if the distance is shorter than $r-\varepsilon$.

Algorithm 1 Query sphere tree S against distance field D

```
1: INPUT: S, D
2: OUTPUT: Set of contacts C
3: Q = {root(S)}
4: while Q ≠ ∅ do
5:    node ← pop_front(Q)
6:    inside ← insideTest(node.p, D)
7:    if inside then
8:       (d, n) ← penetration(node.p, D)
9:       C = C ∪ (node.p, d, n)
10:      if sufficientContacts(C) then
11:         STOP
12:      end if
13:   else
14:      intersects ← sphereTest(node.p, r, D)
15:   end if
16:   if inside OR intersects then
17:      Q = Q ∪ children(node)
18:   end if
19: end while
```

We augment the basic collision detection algorithm with *self-adapting contact selection*. As described in Section 5, we construct the sphere tree in a way that allows adaptive contact selection by simple breadth-first tree traversal, defining a contact constraint whenever we encounter a sphere whose center is inside fragment B, until a sufficient number of constraints is reached.

4. Fragment Distance Field

Given a volumetric meshing of an object A, computed as a preprocess, we propose a fragment distance field data structure that is efficiently stored and updated even upon multiple fractures of the object. This data structure stores an approximate interior distance field of all fragments created by the fractures, using a precomputed volumetric mesh, without any remeshing. Moreover, we exploit the connectivity of the mesh to compute approximate distances in a very fast manner using a front propagation approach.

In this section, we first describe the distance field data structure and its run-time update, and then we describe how it is used to answer the three elementary queries outlined in the previous section.

4.1. Mesh-Based Interior Distance

Our distance field data structure is motivated by features of fracture simulation algorithms. First, the fragments resulting from a brittle fracture define an exact partition of the

original object. Therefore, each point of the original object needs to store only one interior distance value even after multiple fractures. And second, popular approaches for fracture simulation use a volumetric mesh to compute an elastic deformation field and guide the propagation of crack surfaces [OH99, MDMJ01]. The virtual node algorithm and subsequent adaptations [MBF04, PO09, GMD12] limit the resolution of newly created fragments, forcing them to include one node of the original mesh. We exploit this feature and store one interior distance value at each node of the original volumetric mesh.

In our implementation, we have used a tetrahedral mesh as volumetric mesh for storing the fragment distance field. Specifically, each node of the mesh stores:

- f: an identifier of the fragment that contains the node.
- d: a value that approximates the shortest distance to the surface of fragment f.
- $\mathbf{n}$: a unit vector that approximates the direction from the node to the closest surface of f.

As a preprocess, we initialize the nodal information using an exact interior distance field.

In addition to nodal information, tetrahedra that are intersected by crack surfaces store exact local representations of those crack surfaces. Following the virtual node approach, each tetrahedral edge may be cut at most once, therefore, the storage requirements are limited to six plane equations.

4.2. Distance Updates upon Fracture

After each fracture event, we locally update the fragment distance field where needed, following a front propagation approach. The run-time computation of the exact distance field is computationally prohibitive, but we propose a fast approximation that fulfills desired properties. It is important to remark that the interior distance of a fragment is used in the computation of penetration depth and contact normal in the collision detection Algorithm 1. The amount of penetration depth is used by the drift correction algorithm during collision response, and the contact normal is used for the definition of non-penetration contact constraints. Distances need not be accurate, but they must grow monotonically in the interior of an object, locally approximate Euclidean distance, and converge to the true distance as we get close to the surface of the object. Normal directions, on the other hand, must point outward of the object, and should vary smoothly to avoid competing contact constraints for nearby points. It turns out that the algorithm for consistent penetration depth computation of Heidelberger et al. [HTK*04] fulfills exactly these properties, hence we have adapted this algorithm for interior distance field approximation.

Next, we summarize the application of Heidelberger's algorithm to our problem. When an object A suffers a fracture, we first visit all tetrahedra intersected by the newly created crack surfaces, and initialize distance field information

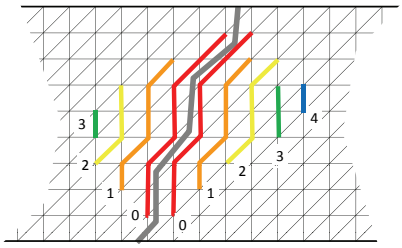

Figure 2: *Illustration of the front propagation algorithm for interior distance field computation.*

at their nodes. This implies assigning a fragment identifier f, and computing a distance d and a direction $\mathbf{n}$, based on the exact surface information stored at the tetrahedra. For each fragment, we initialize a front with the visited nodes. Then, we iterate front propagation steps on the graph defined by tetrahedral edges, until no distances are reduced. Fig. 2 illustrates the front propagation inside a fragment.

If the front propagation reaches a node at position $\mathbf{p}$ in step $i+1$, we compute a distance to the surface as an average propagation of distances from its neighbors reached in step i (denoted as $N_i(\mathbf{p})$), in the following manner:

$$d = \frac{\sum_{j \in N_i(\mathbf{p})} w_j \left(d(\mathbf{p}_j) + \mathbf{n}(\mathbf{p}_j)^T (\mathbf{p}_j - \mathbf{p}) \right)}{\sum_{j \in N_i(\mathbf{p})} w_j}. \quad (1)$$

Following Heidelberger et al., we use as neighbor weights $w_j = 1/\|\mathbf{p}_j - \mathbf{p}\|^2$. If the distance d is shorter than the current value stored at $\mathbf{p}$, we update the distance and add $\mathbf{p}$ to the front at step $i+1$. We also update the direction at $\mathbf{p}$ as the weighted average direction of neighbors reached in step i:

$$\mathbf{n} = \frac{\sum_{j \in N_i(\mathbf{p})} w_j \mathbf{n}(\mathbf{p}_j)}{\sum_{j \in N_i(\mathbf{p})} w_j}, \quad (2)$$

followed by a normalization step.

Fig. 3 shows an accurate interior distance field for an object A, the accurate distance fields of its fragments after a fracture, and our approximate distance fields. The image illustrates the monotonic growth of distances inside the fragments. Our distance field approximation does not require high-quality tetrahedral meshes in practice. In our examples, we used TetGen for mesh generation, with a maximum radius-edge ratio between 1 and 2, and enforcing interior edges to be shorter than twice the length of the longest surface edge.

Even though we have used tetrahedral meshes, our data structure could be extended to other settings, such as hexahedral meshes or even meshless methods. The mesh is used in two ways: (i) Its edges define a graph for the propagation of distances; (ii) Distances can be interpolated inside mesh elements. On hexahedral meshes, the graph may be constructed using the edges of the mesh or adding other con-

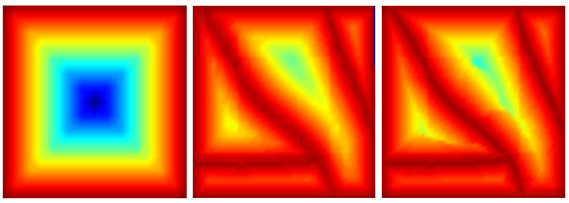

Figure 3: *From left to right: interior distance field of a 2D object, distance fields of its four fragments after fracture, and our approximate distance fields computed using a fast propagation method.*

nections, and interpolation can also be defined inside mesh elements. On meshless methods, a graph may be constructed using neighboring particle information which is easily updated upon fracture events [SOG06], and interpolation can be defined based on neighboring nodes [MKN*04].

4.3. Inside-Outside Query: insideTest($\mathbf{p}, D(f)$)

As a preprocess, we build a k-d tree with the tetrahedra of the mesh. To decide whether a point $\mathbf{p}$ is inside a fragment f, we first use the k-d tree to retrieve the tetrahedron that encloses $\mathbf{p}$. If all nodes of the tetrahedron are in the same fragment, then the query is trivially answered. If only some nodes are in fragment f, then we use the exact local representation of the crack surfaces to answer the inside-outside query.

4.4. Penetration Depth Query: penetration($\mathbf{p}, D(f)$)

If the tetrahedron containing a point $\mathbf{p}$ is intersected by crack surfaces, we use the exact local surface information to compute the penetration depth and direction to the surface of fragment f. If the tetrahedron is completely inside the fragment, then we use the fragment distance field. In particular, we use as neighbors $N_i(\mathbf{p})$ the four nodes of the tetrahedron, and we apply Eq. (1) to compute the distance to the surface of f, and Eq. (2) to compute the penetration direction.

Close to original surfaces of the object, where fracture does not modify distances, it is possible to obtain more accurate penetration information in a simple manner. As a preprocess, we compute a distance field on a dense regular grid. This regular-grid distance field is used for the initialization of the fragment distance field at nodal positions, but we also query it at run-time. We simply use the minimum of the distances returned by the (precomputed) regular-grid distance field and the (dynamically updated) fragment distance field.

4.5. Sphere Intersection Query: sphereTest($\mathbf{p}, r, D(f)$)

The fragment distance field stores only interior distance information for the fragments. When the query point $\mathbf{p}$ is outside fragment f, the fragment distance field provides the distance d to the surface of some other fragment. This distance d is a lower estimate of the distance to f, and can be used for culling in Algorithm 1 if it is larger than the radius r of the sphere. To handle far fragments, we use the largest between d and the distance to a bounding box of fragment f.

The procedure described above performs well in most cases, but fails for large non-convex fragments surrounded by small fragments, returning largely underestimated distances that produce little culling. We provide a less conservative solution for such situations. As a preprocess, we construct a multi-level grid on every object, and register pointers from the tetrahedra to their occupied cells. Every grid cell stores a bit mask indicating whether it contains each and every fracture fragment. Upon a fracture event, we traverse the tetrahedra of new fragments, mark the bit masks of their occupied cells, and then we perform a bottom-up update of the multi-level grid by simple logical AND operations. To test a sphere for intersection, we query the grid level with cell size immediately larger than $2r$. The sphere can be culled if none of the eight cells joining at the grid point closest to $\mathbf{p}$ contains fragment f.

5. Fracturable Adaptive Sphere Tree

Our novel sphere tree data structure is motivated by two requirements. First, to reduce the cost of both collision detection and response, in particular at collision-intensive fracture events, we seek a sphere tree data structure suitable for adaptive collision detection. We construct a sphere tree by optimizing at each level the coverage of the fragment, in a way similar to the point-shell hierarchy of Barbič and James [BJ08]. In this way, we achieve good contact sampling through simple breadth-first traversal of the sphere tree. Second, the data structure should allow very fast updates upon fracture events. We construct the sphere tree by covering both the surface and the interior of an original object. Prior to fracture, interior parts are easily culled and produce almost no overhead. After they get exposed by fracture, on the other hand, they are quickly accessed during tree traversal.

Next, we describe the ordering of points and the construction of our fracturable adaptive sphere tree, the procedure for updating the sphere tree upon fracture, and the execution of self-adapting collision detection.

5.1. Ordering and Construction of the Sphere Tree

We build a sphere tree on a set of points $P = \{\mathbf{p}_i\}$ representing an object A. As discussed in Section 4.1, we assume that the fracture algorithm relies on a volumetric mesh associated with the object. To anticipate fracture events, the set of points consists of the union of the surface vertices and the nodes of the volumetric mesh. The full role of interior nodes during tree updates will be explained in Section 5.2.

For good adaptive collision detection during tree traversal, we seek a good sampling of the set P on every level

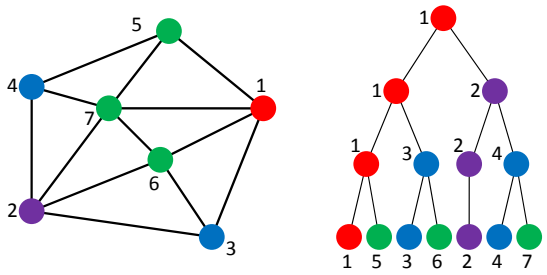

Figure 4: *A 2D polygon with surface vertices and interior nodes (left) and its sphere tree (right). The points are numbered according to maximum distance ordering, and colored according to the level in which they are added to the tree.*

of the tree. This can be achieved by selecting most distant points, which would produce a good coverage of the object. Barbič and James [BJ08] seed random points for each level of the tree, and achieve good coverage thanks to a relaxation algorithm. Although their approach might be adapted to our setting, we seek two additional features: our set of points includes interior points in addition to surface points, and the sampling retains the original surface vertices and interior nodes, to later accommodate fracture updates. In addition, we assume meshes sampled in a semi-regular way.

We achieve good coverage of the object at each level through *maximum distance ordering* of the points P. We initialize an ordered list L_2 with the two furthest points. Then, given the ordered list with m points, L_m, we append the point that is furthest from its closest point in L_m, i.e., $L_{m+1} = (L_m, \mathbf{p}^*)$, with $\mathbf{p}^* = \arg\max_{\mathbf{p}_i \notin P_m} \min_{\mathbf{p}_j \in P_m} \|\mathbf{p}_i - \mathbf{p}_j\|$.

Given the full ordered list, level l of a sphere tree, with 2^l nodes, is trivially constructed by selecting the first 2^l points in the list. Then, level $l+1$ is constructed using those same 2^l nodes and the following 2^l nodes in the list. We set as parent of a node in level $l+1$ its closest node in level l, just like Barbič and James. This heuristic groups nodes based on proximity and increases the chances for pruning during runtime queries. Fig. 4 shows a 2D example with the maximum distance ordering and the tree construction. The sphere tree construction is a preprocess, and we have followed an unoptimized $O(n^2)$ implementation based on pair-wise distance computation, but accelerations are possible.

Each node of the tree must store the sphere center (i.e., the point's position $\mathbf{p}$) and radius. For a node with center at $\mathbf{p}$, we precompute the radius as the distance to its furthest descendant. Each point may be present at multiple levels in the tree (but with different radii). We define a contact constraint only the first time the point is queried in Algorithm 1, and we cache its inside-outside status for subsequent queries down the tree. Choosing the point $\mathbf{p}$ as the center of the sphere does not yield optimally tight spheres. We tried approaches that produce tighter spheres with better culling, but the overall query times were worse as we could not exploit caching.

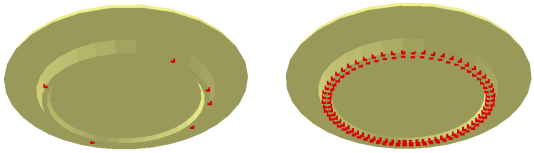

Figure 5: *Rolling plate on a transparent ground, with contacts output by (left) our self-adapting collision detection (up to 6), and (right) full collision detection (up to 128). See the accompanying video.*

5.2. Tree Updates upon Fracture

We restructure the sphere trees through simple local modifications of parent-child relationships. Given a node belonging to a fragment f_i, if its parent belongs to a different fragment f_j, we make the node a child of its closest ancestor in f_i. If it has no ancestor in f_i, the node becomes a root for f_i. A direct implication of this decision is that sphere trees may become forests of several trees after fracture events.

When an edge of the volumetric mesh is cut by a crack surface, two new points need to be added to two different fragments. Recall that we assume fracture algorithms adopting the virtual node approach [MBF04], and then each edge is cut at most once. Given a new point $\mathbf{p}_i$, we make it a child of its neighboring original point $\mathbf{p}_j$. Note that $\mathbf{p}_i$ is a surface point, and we follow an insertion approach that tries to place $\mathbf{p}_i$ high in the hierarchy.

The insertion of new points also requires the modification of sphere radii, as they may no longer be conservative. When $\mathbf{p}_i$ is added, we check if the radius of its parent $\mathbf{p}_j$ is shorter than $\|\mathbf{p}_i - \mathbf{p}_j\|$, and we update it accordingly. We also propagate updates up in the tree if needed.

Compared to full tree recomputation, our fast tree update offers a much more efficient solution at fracture events, and a good compromise for subsequent simulation frames. In simulations where fracture events are distributed over time, our approach could be extended with full tree recomputation as a parallel task, followed by data structure swapping.

5.3. Self-Adapting Collision Detection

The fragment distance field and the fracturable sphere tree enable fast queries and fast data structure updates upon fracture. However, in situations with many penetrating points or with parallel surfaces in close proximity, the cost of collision detection is inevitably high, and collision response is affected by the large number of contacts. We have designed a self-adapting collision detection algorithm that produces a reduced set of contact constraints, and at the same time guarantees the absence of penetration (at the final stable configuration). Our algorithm relies on a velocity-level constraint-

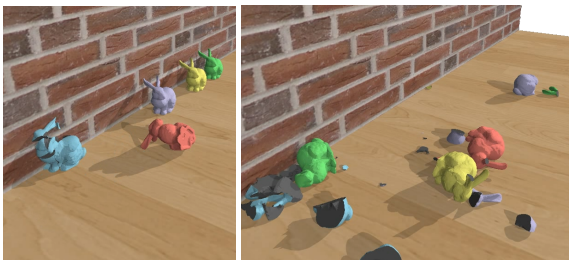

Figure 6: *The user manipulates bunnies interactively with the mouse, producing fractures and collisions. The complete simulation runs at* 2*ms per time step on average, with a maximum of* 10*ms.*

based contact solver plus drift correction, the gold standard solution in rigid body engines in video games.

During breadth-first traversal of the sphere tree, we may output contact constraints high in the hierarchy as outlined in Algorithm 1. Thanks to the good sampling provided by the maximal distance ordering, a few of the first encountered contacts are probably sufficient for the velocity-level constraint-based solver, while further contacts become redundant. We initialize a collision query between two fragments f_i and f_j by setting a maximum number of contacts m (8 in our experiments), and if this number is reached we simply interrupt the query. Drift correction quickly resolves the contacts that have been detected, but if this number is m, then other contacts may have been missed. In that case, we increment $m \leftarrow m+1$, and continue the sphere tree traversal with a negative tolerance $-\varepsilon$ (in our experiments $\varepsilon = 0.2\%$ of the object radius), i.e., we search for contacts that penetrate further than ε. Effectively, with this approach collision detection self-adapts until the number of contacts guarantees non-penetration up to a tolerance ε on a stable configuration. Fig. 5 compares the number of contacts for a $5,392$-triangle plate rolling on a transparent ground with our approach vs. full collision detection. Self-adapting collision detection requires at most 6 contacts, while full collision detection outputs up to 128 contacts. In self-adapting collision detection, adaptivity could also be guided by error metrics of collision response, but existing approaches do not address the complex interactions of constraint-based collision response.

We found that, to be effective at fracture events, self-adapting collision detection requires a small positive detection tolerance ε, i.e., we output contacts closer than a small distance ε. The reason is that the tree traversal stops only when m contacts are output, and parallel surfaces just about to touch would allow little culling but produce no contacts.

6. Experiments and Results

We evaluated our approach on 5 scenarios: (i) a piggy bank dropped on the ground (Fig. 7), (ii) 27 bunnies dropped at different times (Fig. 10), (iii) 32 bricks crashed against the ground (Fig. 11), (iv) an interactive scenario where the user drops balls on 4 plates placed on a shelf (Fig. 1), and (v) another interactive scenario where the user manipulates and fractures 5 bunnies (Fig. 6). The sizes of the surface and volumetric meshes of the different objects are summarized in Table 1. Our collision detection algorithm has been integrated with the rigid body engine of Havok, and we have used a fast fracture simulation method based on modal analysis [GMD12]. The 'freezing' utility of Havok Physics was deactivated in all experiments, for better analysis. All experiments were executed on a 3.4GHz Intel i7-2600 processor with 8GB of RAM, using only one core.

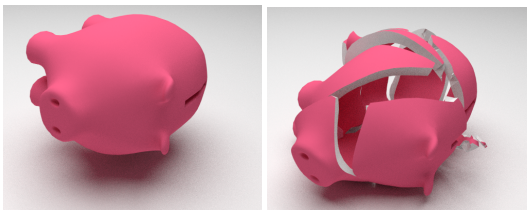

Figure 7: *Piggy bank used for comparisons and analysis.*

Object	# Triangles	# Tetrahedra	# Points
Piggy bank	9,722	20,807	5,870
Bunny	7,940	18,767	5,089
Brick	468	594	224
Plate	5,392	8,617	2,711
Shelf	4652	10,200	2,989

Table 1: *Number of triangles, tetrahedra, and points (including surface vertices and interior points) for the different objects used in the experiments.*

Tables 2 and 3 report various simulation statistics and timings for the 5 benchmarks. The 'piggy bank', 'bricks', 'plates', and 'interactive bunnies' benchmarks are all real-time, including dynamics simulation, fracture simulation, collision detection, and collision response. The 'drop bunnies' scenario, on the other hand, was executed with a sub-frame time step (5ms) to illustrate robust contact handling with small fragments and high impact velocities.

Fig. 12 shows plots of timings and simulation statistics for the 'drop bunnies' and 'bricks' scenarios. In both examples, the cost of collision detection grows steadily as more objects are dropped. However, we can draw the important observation that collision detection does not suffer noticeable spikes at fracture events, despite the large number of colliding points, thanks to our self-adapting contact selection. Both scenarios show computational peaks at fracture events due to the cost of fracture computation. The cost of data structure updates was always smaller than the cost of

Scenario	# Triangles Original/Fractured	# Fracture Fragments	# Output points Average (Max)	# Colliding points Average (Max)
Piggy bank	9,734 / 17,622	14	93 (463)	290 (4,173)
Drop bunnies	137,403 / 435,068	166	1,235 (3,245)	4,993 (18,592)
Bricks	15,036 / 40,750	156	671 (1,379)	1,334 (2,700)
Plates	26,268 / 45,762	44	309 (524)	X
Interactive bunnies	39,724 / 51,752	22	326 (470)	X

Table 2: *Simulation statistics for the different scenarios: number of triangles of the scene before and after fracture; total number of fragments; number of contacts selected by collision detection for collision response; and total number of colliding points (not measured in the interactive scenarios).*

Scenario	Time step	Total time Average (Max)	Collision detection Average (Max)	Physics Average (Max)	Update Max	Fracture Max
Piggy bank	15	1.94 (15.2)	1.74 (9.46)	0.14 (0.59)	1.2	12.5
Drop bunnies	5	24.5 (81.5)	23 (70)	1.3 (3.66)	4.4	22
Bricks	30	11.7 (29.5)	10.7 (27.3)	1 (2.46)	1.05	2
Plates	30	7.83 (17.4)	6.78 (12.2)	0.36 (0.7)	0.7	8
Interactive bunnies	15	2 (10.6)	1.64 (4)	0.26 (0.41)	1	9

Table 3: *Break-up of timings for the different scenarios, all given in milliseconds, and showing average and maximum cost per time step: time step size (with frames rendered every 30ms); total cost per time step; time for collision detection queries; time for physics computations, numerical integration, and collision response; time for data structure updates; and time for fracture computations. The last two times are measured only at fracture events.*

fracture computation, and is not showed for clarity (but it is summarized in Table 3).

We have also analyzed the influence of resolution (both for the surface mesh and the interior sampling of the volumetric mesh) on data structure updates and collision detection queries (for the sphere in Fig. 8). The timings for a reference sphere (2.5K triangles and 4K interior points) are: 1.54ms for updates upon fracture, and 1.16ms on average (3.27ms max) for queries. Changing the surface resolution (to 10K triangles), while keeping the interior sampling fixed, timings are: 1.8ms for the update, and 1.37ms on average (4.17ms max) for queries. Changing the interior sampling (to 435 points), while keeping the surface fixed, timings are: 0.46ms for the update, and 0.68ms on average (2.26ms max) for queries.

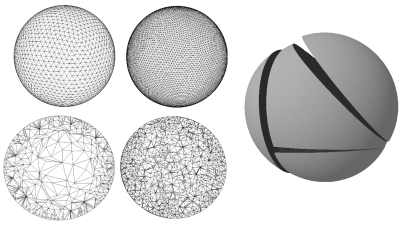

Figure 8: *Sphere used for the analysis of sampling resolution on update and query costs. The top left images show two different samplings of the surface, and the bottom left images show two different samplings of the interior.*

Finally, we have also analyzed the overhead introduced in collision detection queries by our data structures, which trade fast updates upon fracture for not fully optimal culling. Fig. 9 plots several comparisons for the 'piggy bank' scenario in Fig. 7. Our approach updates the distance field (D) and the sphere tree (S) when the piggy bank crashes. We have compared collision detection query times and the number of visited points in the sphere trees, with other combinations where we recompute the exact distance field and/or we recompute the sphere tree for the new surface (with no interior points).

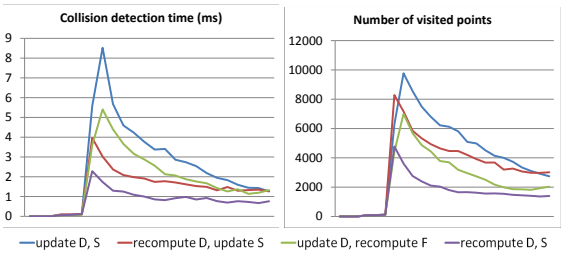

Figure 9: *Comparison of query times and statistics for the 'piggy bank' scene. We compare our fast update of the distance field D and sphere tree S data structures to other combinations that fully recompute an exact distance field and/or a sphere tree of the new surfaces. We achieve similar culling efficiency with much faster updates at fracture events.*

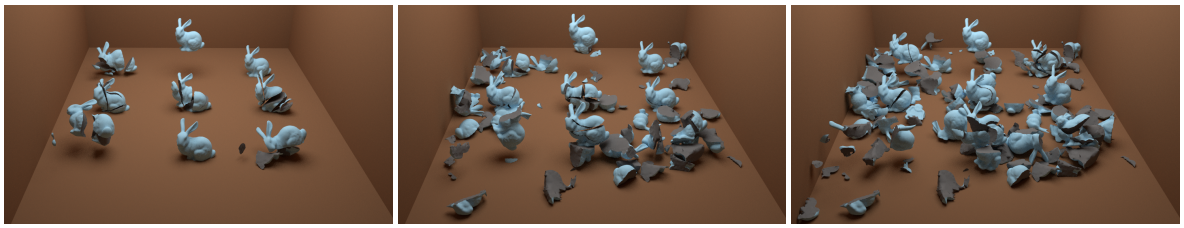

Figure 10: *Bunnies are dropped in three batches and fractured into* 166 *fragments and* 435*K triangles. The complete simulation runs at* 24.5*ms per time step on average, with a maximum of* 81.5*ms.*

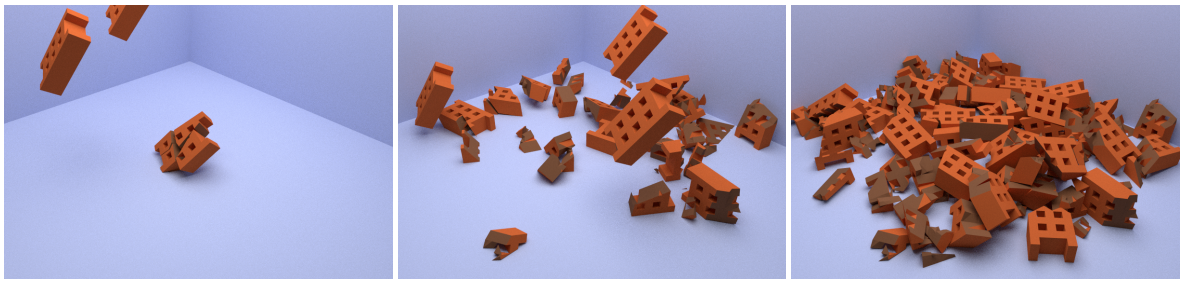

Figure 11: *Real-time demo of crashing bricks, totalling* 156 *fragments and* 40*K triangles. The complete simulation runs at* 11.7*ms per time step on average, with a maximum of* 29.5*ms.*

7. Discussion and Future Work

In this paper, we proposed an efficient solution for collision detection in brittle fracture simulations. Our solution is composed of algorithms that address the two main challenges in such simulations: the update of acceleration data structures upon topology changes, and the efficient computation of contacts between newly created crack surfaces.

Our algorithms demonstrate high performance in challenging scenarios, including real-time user manipulation of fracturing objects, and scenes with hundreds of fragments and tens of thousands of triangles simulated at video game rates. Some limitations remain however, including the possibility to miss collisions with small features and robustness problems under large penetrations. Solving these limitations requires non-trivial extensions to incorporate continuous collision detection.

We envisage other interesting extensions as well. One is the design of parallel versions of our algorithms, to exploit the computing power of graphics processors. Another one is the application of our solutions to penalty-based collision response. The adaptive contact selection was designed for constraint-based response algorithms and may not be trivially adapted, but other components, such as the fragment distance field, may be easily integrated. Yet another interesting extension is handling ductile and/or progressive fracture and elastic deformations. Since our approach already updates data structures at fracture events, it should also be possible to update those data structures as objects deform and fractures progress. Distance fields and sphere trees could also serve for self-collision detection algorithms.

The results of our experiments open promising perspectives for the use of our solutions in real-time applications such as video games and haptic interaction.

Acknowledgements

This research is supported in part by the Spanish Ministry of Economy (TIN2009-07942) and by the European Research Council (ERC-2011-StG-280135 Animetrics).

References

[BBC*11] BAKER M., BIN ZAFAR N., CARLSON M., COUMANS E., CRISWELL B., HARADA T., KNIGHT P.: *Destruction and Dynamics for Film and Game Production*. ACM SIGGRAPH Course Notes, 2011. 1

[BHTF07] BAO Z., HONG J.-M., TERAN J., FEDKIW R.: Fracturing rigid materials. *IEEE Transactions on Visualization and Computer Graphics 13*, 2 (2007), 370–378. 1

[BJ08] BARBIČ J., JAMES D. L.: Six-DoF haptic rendering of contact between geometrically complex reduced deformable models. *IEEE Trans. on Haptics 1*, 1 (2008). 2, 5, 6

[Erl07] ERLEBEN K.: Velocity-based shock propagation for multibody dynamics animation. *ACM Trans. on Graphics 26*, 2 (2007). 2

[FBAF08] FAURE F., BARBIER S., ALLARD J., FALIPOU F.: Image-based collision detection and response between arbitrary volume objects. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2008), pp. 155–162. 3

[FL01] FISHER S., LIN M. C.: Fast penetration depth estimation for elastic bodies using deformed distance fields. *Proc. of IEEE/RSJ Int. Conf. on Intelligent Robots and Systems* (2001). 2

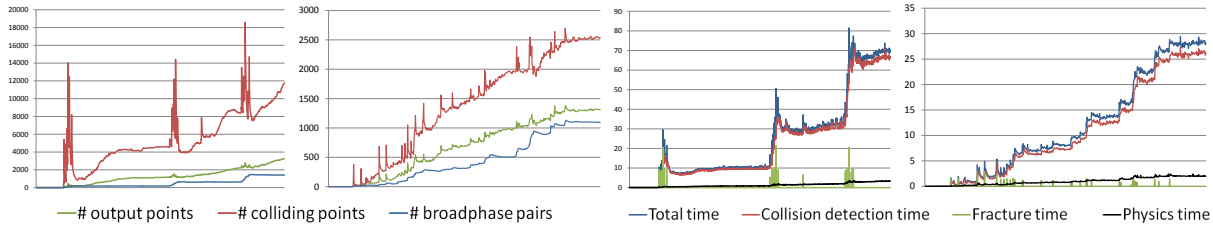

Figure 12: *Plots for the 'drop bunnies' scene (first and third plots) and the 'bricks' scene (second and fourth plots). The two left plots show collision detection statistics: number of points output by adaptive collision detection (green), number of actual colliding points (red), and pairs of bodies output by broad-phase collision detection (blue). The two right plots show timings per time step: total (blue), collision detection (red), fracture computation (green), and physics computations (black). Times to update collision detection data structures were always shorter than fracture computation, and are not included for clarity.*

[FPRJ00] FRISKEN S., PERRY R., ROCKWOOD A., JONES R.: Adaptively sampled distance fields: A general representation of shapes for computer graphics. In *Proc. of ACM SIGGRAPH* (2000), pp. 249–254. 2

[GBF03] GUENDELMAN E., BRIDSON R., FEDKIW R.: Nonconvex rigid bodies with stacking. *Proc. of ACM SIGGRAPH* (2003). 2

[GMD12] GLONDU L., MARCHAL M., DUMONT G.: Real-time simulation of brittle fracture using modal analysis. *IEEE Trans. on Visualization and Computer Graphics* (2012). 1, 4, 7

[HSK*10] HEO J.-P., SEONG J.-K., KIM D., OTADUY M. A., HONG J.-M., TANG M., YOON S.-E.: FASTCD: Fracturing-aware stable collision detection. *Proc. of ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2010). 2

[HTK*04] HEIDELBERGER B., TESCHNER M., KEISER R., MÜELLER M., GROSS M.: Consistent penetration depth estimation for deformable collision response. *Proc. of Vision, Modeling and Visualization* (2004). 2, 4

[Hub96] HUBBARD P. M.: Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics 15*, 3 (1996), 179–210. 2

[KNF04] KIMMERLE S., NESME M., FAURE F.: Hierarchy accelerated stochastic collision detection. *Proc. of Vision, Modeling and Visualization* (2004). 2

[KSJP08] KAUFMAN D. M., SUEDA S., JAMES D. L., PAI D. K.: Staggered projections for frictional contact in multibody systems. *Proc. of ACM SIGGRAPH Asia* (2008). 2

[KSP07] KAUFMAN D. M., SUEDA S., PAI D. K.: Contact trees: Adaptive contact sampling for robust dynamics. *SIGGRAPH, Technical Sketches* (2007). 2

[KZ03] KLEIN J., ZACHMANN G.: ADB-Trees: Controlling the error of time-critical collision detection. *Proc. of Vision, Modeling and Visualization* (2003). 2

[LA06] LARSSON T., AKENINE-MÖLLER T.: A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics 30* (2006). 2

[LGS*09] LAUTERBACH C., GARLAND M., SENGUPTA S., LUEBKE D., MANOCHA D.: Fast bvh construction on gpus. *Proc. of Eurographics* (2009). 3

[LHLK10] LIU F., HARADA T., LEE Y., KIM Y. J.: Real-time collision culling of a million bodies on graphics processing units. *Proc. of ACM SIGGRAPH Asia* (2010). 3

[MBF04] MOLINO N., BAO Z., FEDKIW R.: A virtual node algorithm for changing mesh topology during simulation. *ACM Transactions on Graphics 23*, 3 (2004), 385–392. 4, 6

[MDMJ01] MÜLLER M., DORSEY J., MCMILLAN L., JAGNOW R.: Real-time simulation of deformation and fracture of stiff materials. *Proc. of Eurographics Workshop on Animation and Simulation* (2001). 4

[MKN*04] MÜLLER M., KEISER R., NEALEN A., PAULY M., GROSS M., ALEXA M.: Point based animation of elastic, plastic and melting objects. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2004), pp. 141–151. 5

[MPT99] MCNEELY W. A., PUTERBAUGH K. D., TROY J. J.: Six degrees-of-freedom haptic rendering using voxel sampling. In *Proc. of ACM SIGGRAPH* (1999), pp. 401–408. 2

[OCSG07] OTADUY M. A., CHASSOT O., STEINEMANN D., GROSS M.: Balanced hierarchies for collision detection between fracturing objects. In *Proc. of IEEE Virtual Reality Conference* (2007). 2

[OD99] O'SULLIVAN C., DINGLIANA J.: Real-time collision detection and response using sphere-trees. In *Proc. 15th Spring Conference on Computer Graphics* (1999), pp. 83–92. 2

[OH99] O'BRIEN J. F., HODGINS J. K.: Graphical modeling and animation of brittle fracture. In *Proc. of ACM SIGGRAPH* (1999), pp. 137–146. 1, 4

[OL03] OTADUY M. A., LIN M. C.: CLODs: Dual hierarchies for multiresolution collision detection. In *Proc. Eurographics Symposium on Geometry Processing* (2003), pp. 94–101. 2

[PG94] PALMER I. J., GRIMSDALE R. L.: Collision detection for animation using sphere-trees. *Computer Graphics Forum 14*, 2 (1994), 105–116. 2

[PO09] PARKER E. G., O'BRIEN J. F.: Real-time deformation and fracture in a game environment. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2009), pp. 156–166. 1, 4

[SGGM06] SUD A., GOVINDARAJU N. K., GAYLE R., MANOCHA D.: Interactive 3d distance field computation using linear factorization. *Proc. of ACM Symposium on Interactive 3D Graphics and Games* (2006). 2, 3

[SOG06] STEINEMANN D., OTADUY M. A., GROSS M.: Fast arbitrary splitting of deforming objects. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2006), pp. 63–72. 2, 5

[WZ09] WELLER R., ZACHMANN G.: Inner sphere trees for proximity and penetration queries. *Robotics: Science and Systems* (2009). 2